

# Instruction Sequence Size Complexity of Parity

J.A. Bergstra and C.A. Middelburg

Informatics Institute, Faculty of Science, University of Amsterdam,  
Science Park 904, 1098 XH Amsterdam, the Netherlands

J.A.Bergstra@uva.nl, C.A.Middelburg@uva.nl

**Abstract.** Each Boolean function can be computed by a single-pass instruction sequence that contains only instructions to set and get the content of Boolean registers, forward jump instructions, and a termination instruction. Auxiliary Boolean registers are not necessary for this. In the current paper, we show that, in the case of the parity functions, shorter instruction sequences are possible with the use of an auxiliary Boolean register in the presence of instructions to complement the content of auxiliary Boolean registers. This result supports, in a setting where programs are instruction sequences acting on Boolean registers, a basic intuition behind the storage of auxiliary data, namely the intuition that this makes possible a reduction of the size of a program.

**Keywords:** Boolean function family, instruction sequence size, non-uniform complexity measure, parity function.

*1998 ACM Computing Classification:* F.1.1, F.1.3.

## 1 Introduction

In [1], we presented an approach to computational complexity in which algorithmic problems are viewed as families of functions that consist of an  $n$ -ary Boolean function for each natural number  $n$  and the complexity of such problems is assessed in terms of the length of finite single-pass instruction sequences acting on Boolean registers that compute the members of these families. The instruction sequences concerned contain only instructions to set and get the content of Boolean registers, forward jump instructions, and a termination instruction. Moreover, each Boolean register used serves as either input register, output register or auxiliary register.

Auxiliary Boolean registers are not needed to compute Boolean functions. The question whether shorter instruction sequences are possible with the use of auxiliary Boolean registers was not answered in [1]. In the current paper, we show that, in the case of the parity functions, shorter instruction sequences are possible with the use of an auxiliary Boolean register provided the instruction set is extended with instructions to complement the content of auxiliary Boolean registers. The parity function of arity  $n$  is the function from  $\{0,1\}^n$  to  $\{0,1\}$  whose value at  $b_1, \dots, b_n$  is 1 if and only if the number of 1's in  $b_1, \dots, b_n$  is odd.

The parity functions have a well-known practical application. If we append to a bit string the value of the parity function for this bit string, the appended bit is

called the parity bit. After appending the parity bit, the number of times that 1 occurs is always even. A test for this property is called a parity check. Appending parity bits and performing parity checks are used in many techniques to detect errors in transmission of binary data (for an overview, see e.g. [2, Part I]).

In theoretical computer science, the complexity of the parity functions has been extensively studied (see e.g. [3,4,5,6,7,8]). All these studies have been carried out in the setting of Boolean circuits. Moreover, to our knowledge, none of these studies is concerned with an issue comparable to the one considered in this paper, namely the issue whether, in the case of computing the parity functions, shorter instruction sequences are possible with the use of auxiliary Boolean registers than without the use of auxiliary Boolean registers.

Our results concerning this issue can be paraphrased as follows: in the presence of instructions to complement the content of auxiliary Boolean registers, (i) the parity function of arity  $n$  can be computed by an instruction sequence without the use of an auxiliary Boolean register where the length of the instruction sequence is not more than  $5n - 2$  but more than  $2n + 3$  and (ii) the parity function of arity  $n$  can be computed by an instruction sequence with the use of one auxiliary Boolean register where the length of the instruction sequence is not more than  $2n + 3$ . Consequently, smaller instruction sequences are possible with the use of an auxiliary Boolean register.

In our results, the presence of instructions to complement the content of auxiliary Boolean registers is assumed. In [1], instruction sequences that contain these instructions were not considered. Since the results from that paper, with the exception of one auxiliary result, are concerned with upper bounds of instruction sequence size complexity, they go through if instruction sequences may contain instructions to complement the content of auxiliary Boolean registers as well. However, when doing some other kinds of complexity analysis, e.g. when proving lower bounds of instruction sequence size complexity, the instructions assumed to be present can matter. In [9], we consider all instructions for Boolean registers that are possible in the setting in which the work presented in this paper is carried out and study the effect of their presence on instruction sequence size.

We give in this paper, in a setting where programs are instruction sequences acting on Boolean registers, an example where the storage of auxiliary data makes possible a reduction of program size. We cannot find any other work supporting the intuitively evident fact that it must happen quite often that the storage of auxiliary data leads to a reduction of program size.

Since Boolean registers can be looked upon as program variables, the work on the transformation of programs to structured programs by coding previous flow of control into auxiliary Boolean variables that is presented in [10,11] can be considered related work. However, we consider that work only moderately related because it concerns the increase of the degree of structuredness of programs instead of the decrease of the size of programs. All the work on program correctness in which auxiliary program variables play a role, starting with [12] (see [13] for an overview), can be considered somewhat related as well. However,

we consider that work only loosely related because it restricts the use of auxiliary program variables to the left-hand sides of assignments whereas, in this paper, the use of auxiliary Boolean registers is not restricted at all.

The work presented in this paper is carried out in the setting of PGA (ProGram Algebra). PGA is an algebraic theory of single-pass instruction sequences that was taken as the basis of an approach to the semantics of programming languages introduced in [14]. As a continuation of the work presented in [14], (i) the notion of an instruction sequence was subjected to systematic and precise analysis and (ii) issues relating to diverse subjects in computer science and computer engineering were rigorously investigated in the setting of PGA. The subjects concerned include programming language expressiveness, computability, computational complexity, algorithm efficiency, algorithmic equivalence of programs, program verification, program compactness, probabilistic programming, and micro-architecture. For a comprehensive survey of a large part of this work, see [15]. An overview of all the work done to date in the setting of PGA and some open questions originating from that work can be found at <http://instructionsequence.wordpress.com>.

This paper is organized as follows. First, we present the preliminaries on instruction sequences and complexity classes based on them that are needed in the rest of the paper (Section 2). Next, we describe how the parity functions can be computed by instruction sequences without the use of auxiliary Boolean registers and with the use of auxiliary Boolean registers (Section 3). Then, we show that the smaller lengths of the instruction sequences in the latter case cannot be obtained without the use of auxiliary Boolean registers (Section 4). Finally, we make some concluding remarks (Section 5).

The preliminaries to the work presented in this paper include excerpts from the preliminaries to the work presented in [1]. The preliminaries include a brief summary of PGA. A comprehensive introduction to PGA, including examples, can among other things be found in [15].

## 2 Preliminaries: Instruction Sequences and Complexity

In this section, we present a brief outline of PGA (ProGram Algebra), the particular fragment and instantiation of it that is used in this paper, and the kind of complexity classes considered in this paper. A mathematically precise treatment for the case without instructions to complement the content of Boolean registers can be found in [1].

The starting-point of PGA is the simple and appealing perception of a sequential program as a single-pass instruction sequence, i.e. a finite or infinite sequence of instructions of which each instruction is executed at most once and can be dropped after it has been executed or jumped over.

It is assumed that a fixed but arbitrary set  $\mathfrak{A}$  of *basic instructions* has been given. The intuition is that the execution of a basic instruction may modify a state and produces a reply at its completion. The possible replies are 0 and 1. The actual reply is generally state-dependent. Therefore, successive executions

of the same basic instruction may produce different replies. The set  $\mathfrak{A}$  is the basis for the set of instructions that may occur in the instruction sequences considered in PGA. The elements of the latter set are called *primitive instructions*. There are five kinds of primitive instructions, which are listed below:

- for each  $a \in \mathfrak{A}$ , a *plain basic instruction*  $a$ ;
- for each  $a \in \mathfrak{A}$ , a *positive test instruction*  $+a$ ;
- for each  $a \in \mathfrak{A}$ , a *negative test instruction*  $-a$ ;
- for each  $l \in \mathbb{N}$ , a *forward jump instruction*  $\#l$ ;
- a *termination instruction*  $!$ .

We write  $\mathfrak{I}$  for the set of all primitive instructions.

On execution of an instruction sequence, these primitive instructions have the following effects:

- the effect of a positive test instruction  $+a$  is that basic instruction  $a$  is executed and execution proceeds with the next primitive instruction if 1 is produced and otherwise the next primitive instruction is skipped and execution proceeds with the primitive instruction following the skipped one — if there is no primitive instruction to proceed with, inaction occurs;
- the effect of a negative test instruction  $-a$  is the same as the effect of  $+a$ , but with the role of the value produced reversed;
- the effect of a plain basic instruction  $a$  is the same as the effect of  $+a$ , but execution always proceeds as if 1 is produced;
- the effect of a forward jump instruction  $\#l$  is that execution proceeds with the  $l$ th next primitive instruction of the instruction sequence concerned — if  $l$  equals 0 or there is no primitive instruction to proceed with, inaction occurs;
- the effect of the termination instruction  $!$  is that execution terminates.

To build terms, PGA has a constant for each primitive instruction and two operators. These operators are: the binary concatenation operator  $;$  and the unary repetition operator  $^\omega$ . We use the notation  $\dot{\bigcup}_{i=k}^n P_i$ , where  $k \leq n$  and  $P_k, \dots, P_n$  are PGA terms, for the PGA term  $P_k ; \dots ; P_n$ .

The instruction sequences that concern us in the remainder of this paper are the finite ones, i.e. the ones that can be denoted by closed PGA terms in which the repetition operator does not occur. Moreover, the basic instructions that concern us are instructions to set and get the content of Boolean registers. More precisely, we take the set

$$\begin{aligned} & \{\text{in}:i.\text{get} \mid i \in \mathbb{N}_1\} \cup \{\text{out.set}:b \mid b \in \{0, 1\}\} \\ & \cup \{\text{aux}:i.\text{get} \mid i \in \mathbb{N}_1\} \cup \{\text{aux}:i.\text{set}:b \mid i \in \mathbb{N}_1 \wedge b \in \{0, 1\}\} \cup \{\text{aux}:i.\text{com} \mid i \in \mathbb{N}_1\} \end{aligned}$$

as the set  $\mathfrak{A}$  of basic instructions.<sup>1</sup>

Each basic instruction consists of two parts separated by a dot. The part on the left-hand side of the dot plays the role of the name of a Boolean register and

<sup>1</sup> We write  $\mathbb{N}_1$  for the set  $\{n \in \mathbb{N} \mid n \geq 1\}$  of positive natural numbers.

the part on the right-hand side of the dot plays the role of a command to be carried out on the named Boolean register. The names are employed as follows:

- for each  $i \in \mathbb{N}_1$ ,  $\text{in}:i$  serves as the name of the Boolean register that is used as  $i$ th input register in instruction sequences;
- **out** serves as the name of the Boolean register that is used as output register in instruction sequences;
- for each  $i \in \mathbb{N}_1$ ,  $\text{aux}:i$  serves as the name of the Boolean register that is used as  $i$ th auxiliary register in instruction sequences.

On execution of a basic instruction, the commands have the following effects:

- the effect of **get** is that nothing changes and the reply is the content of the named Boolean register;
- the effect of **set:0** is that the content of the named Boolean register becomes 0 and the reply is 0;
- the effect of **set:1** is that the content of the named Boolean register becomes 1 and the reply is 1;
- the effect of **com** is that the content of the named Boolean register is complemented and the reply is the complemented content.

We will write  $\text{IS}_{\text{br}}$  for the set of all instruction sequences that can be denoted by a closed PGA term in which the repetition operator does not occur in the case that  $\mathfrak{A}$  is taken as specified above. For each  $k \in \mathbb{N}$ , we will write  $\text{IS}_{\text{br}}^k$  for the set of all instruction sequences from  $\text{IS}_{\text{br}}$  in which primitive instructions of the forms  $\text{aux}:i.c$ ,  $+\text{aux}:i.c$  and  $-\text{aux}:i.c$  with  $i > k$  do not occur. Moreover, we will write  $\text{length}(X)$ , where  $X \in \text{IS}_{\text{br}}$ , for the length of  $X$ .

$\text{IS}_{\text{br}}$  is the set of all instruction sequences that matter to the kind of complexity classes which will be introduced below.  $\text{IS}_{\text{br}}^0$  is the set of all instruction sequences from  $\text{IS}_{\text{br}}$  in which no auxiliary registers are used.

Let  $n \in \mathbb{N}$ , let  $f : \{0, 1\}^n \rightarrow \{0, 1\}$ , and let  $X \in \text{IS}_{\text{br}}$ . Then  $X$  *computes*  $f$  if there exists a  $k \in \mathbb{N}$  such that, for all  $b_1, \dots, b_n \in \{0, 1\}$ , on execution of  $X$  in an environment with input registers  $\text{in}:1, \dots, \text{in}:n$ , output register **out**, and auxiliary registers  $\text{aux}:1, \dots, \text{aux}:k$ , if

- for each  $i \in \{1, \dots, n\}$ , the content of register  $\text{in}:i$  is  $b_i$  when execution starts;
- the content of register **out** is 0 when execution starts;
- for each  $i \in \{1, \dots, k\}$ , the content of register  $\text{aux}:i$  is 0 when execution starts;

then the content of register **out** is  $f(b_1, \dots, b_n)$  when execution terminates.

A *Boolean function family* is an infinite sequence  $\langle f_n \rangle_{n \in \mathbb{N}}$  of functions, where  $f_n$  is an  $n$ -ary Boolean function for each  $n \in \mathbb{N}$ .

Let  $\text{IS} \subseteq \text{IS}_{\text{br}}$  and  $F \subseteq \{h \mid h : \mathbb{N} \rightarrow \mathbb{N}\}$ . Then  $\text{IS} \setminus F$  is the class of all Boolean function families  $\langle f_n \rangle_{n \in \mathbb{N}}$  for which there exists an  $h \in F$  such that, for all  $n \in \mathbb{N}$ , there exists an  $X \in \text{IS}$  such that  $X$  computes  $f_n$  and  $\text{length}(X) \leq h(n)$ . We will use the notation  $\text{IS} \setminus B(f(n))$ , where  $\text{IS} \subseteq \text{IS}_{\text{br}}$  and  $f : \mathbb{N} \rightarrow \mathbb{N}$ , for  $\text{IS} \setminus \{h \mid h : \mathbb{N} \rightarrow \mathbb{N} \wedge \exists m \in \mathbb{N} \bullet (\forall n \in \mathbb{N} \bullet (n \geq m \Rightarrow h(n) \leq f(n)))\}$ .

In [1], it is proved that  $\text{IS}_{\text{br}} \setminus \text{poly}$  coincides with  $\text{P/poly}$ .<sup>2</sup>

<sup>2</sup> As usual,  $\text{poly}$  stands for  $\{h \mid h : \mathbb{N} \rightarrow \mathbb{N} \wedge h \text{ is polynomial}\}$ .

### 3 Computing Parity Functions by Instruction Sequences

In this section, we describe how the parity functions can be computed by instruction sequences in two distinct cases: (i) the case where they are computed without the use of auxiliary Boolean registers and (ii) the case where they are computed with the use of a single auxiliary Boolean register.

The  $n$ -ary parity function  $\text{PAR}_n : \{0, 1\}^n \rightarrow \{0, 1\}$  is defined by

$$\text{PAR}_n(b_1, \dots, b_n) = 1 \quad \text{iff} \quad \text{the number of 1's in } b_1, \dots, b_n \text{ is odd.}$$

We write  $\text{PAR}$  for the Boolean function family  $\langle \text{PAR}_n \rangle_{n \in \mathbb{N}}$ .

We begin with defining instruction sequences which are intended to compute the parity functions without the use of auxiliary Boolean registers. We define instruction sequences  $\text{PARIS}_0^0$  and  $\text{PARIS}_1^0$  as follows:

$$\text{PARIS}_0^0 = !, \quad \text{PARIS}_1^0 = +\text{in}:1.\text{get}; \text{out.set}:1; !$$

and we uniformly define instruction sequences  $\text{PARIS}_n^0$  for  $n \geq 2$  as follows:

$$\text{PARIS}_n^0 = +\text{in}:1.\text{get}; \bigcirc_{i=2}^n (\#4; +\text{in}:i.\text{get}; \#3; \#3; -\text{in}:i.\text{get}); \text{out.set}:1; !.$$

The instruction sequences defined above compute the parity functions. It is not immediately clear why this is the case. The following remark can be of help in understanding it. The contents of  $\text{in}:1, \dots, \text{in}:n$  are read in that order by means of test instructions from the (unfolded) instruction sequence. This happens in such a way that a register is read by means of a positive test instruction if the value of the parity function for the prefix of the bit string that has been dealt with before is 0 and a register is read by means of a negative test instruction if the value of the parity function for the prefix of the bit string that has been dealt with before is 1.

**Proposition 1.** *For each  $n \in \mathbb{N}$ ,  $\text{PARIS}_n^0$  computes  $\text{PAR}_n$ .*

*Proof.* For  $n < 2$ , the proof is trivial. We prove that  $\text{PARIS}_n^0$  computes  $\text{PAR}_n$  for all  $n \geq 2$  by induction on  $n$ . The basis step consists of proving that  $\text{PARIS}_2^0$  computes  $\text{PAR}_2$ . This follows easily by an exhaustive case distinction over the contents of  $\text{in}:1$  and  $\text{in}:2$ . The inductive step is proved in the following way. It follows directly from the induction hypothesis that, after the  $\#4; +\text{in}:i.\text{get}; \#3; \#3; -\text{in}:i.\text{get}$  has been executed  $n$  times, execution proceeds with the next instruction if the number of 1's in the contents of  $\text{in}:1, \dots, \text{in}:n$  is odd and otherwise the next instruction is skipped and execution proceeds with the instruction following the skipped one. From this, it follows easily by an exhaustive case distinction over the content of  $\text{in}:n+1$  that  $\text{PARIS}_{n+1}^0$  computes  $\text{PAR}_{n+1}$ .  $\square$

Because the instruction sequences defined above compute the parity functions, we have a first result about the complexity of  $\text{PAR}$ .

**Theorem 1.**  $\text{PAR} \in \text{IS}_{\text{br}}^0 \setminus B(5 \cdot n - 2)$ .

*Proof.* By simple calculations, we obtain that  $\text{length}(\text{PARIS}_0^0) = 1$  and, for each  $n > 0$ ,  $\text{length}(\text{PARIS}_n^0) = 5 \cdot n - 2$ . From this and Proposition 1, it follows immediately that  $\text{PAR} \in \text{IS}_{\text{br}}^0 \setminus B(5 \cdot n - 2)$ .  $\square$

We go on with defining instruction sequences which are intended to compute the parity functions with the use of a single auxiliary Boolean register. We define an instruction sequence  $\text{PARIS}_0^1$  as follows:

$$\text{PARIS}_0^1 = !$$

and we uniformly define instruction sequences  $\text{PARIS}_n^1$  for  $n \geq 1$  as follows:

$$\text{PARIS}_n^1 = ;_{i=1}^n (+\text{in}:i.\text{get} ; \text{aux}:1.\text{com}) ; +\text{aux}:1.\text{get} ; \text{out.set}:1 ; !.$$

The instruction sequences defined above compute the parity functions as well. At each point of the execution of  $\text{PARIS}_n^1$ ,  $\text{aux}:1$  contains the value of the parity function for the prefix of the bit string that has been dealt with at that point.

**Proposition 2.** For each  $n \in \mathbb{N}$ ,  $\text{PARIS}_n^1$  computes  $\text{PAR}_n$ .

*Proof.* For  $n < 1$ , the proof is trivial. We prove that  $\text{PARIS}_n^1$  computes  $\text{PAR}_n$  for all  $n \geq 1$  by induction on  $n$ . The basis step consists of proving that  $\text{PARIS}_1^1$  computes  $\text{PAR}_1$ . This follows easily by an exhaustive case distinction over the content of  $\text{in}:1$ . The inductive step is proved in the following way. It follows directly from the induction hypothesis that, after the  $+\text{in}:i.\text{get} ; \text{aux}:1.\text{com}$  has been executed  $n$  times, the content of  $\text{aux}:1$  is 1 if the number of 1's in the contents of  $\text{in}:1, \dots, \text{in}:n$  is odd and otherwise the content of  $\text{aux}:1$  is 0. From this, it follows easily by an exhaustive case distinction over the content of  $\text{in}:n+1$  that  $\text{PARIS}_{n+1}^1$  computes  $\text{PAR}_{n+1}$ .  $\square$

Because the instruction sequences defined above compute the parity functions as well, we have a second result about the complexity of  $\text{PAR}$ .

**Theorem 2.**  $\text{PAR} \in \text{IS}_{\text{br}}^1 \setminus B(2 \cdot n + 3)$ .

*Proof.* By simple calculations, we obtain that  $\text{length}(\text{PARIS}_0^1) = 1$  and, for each  $n > 0$ ,  $\text{length}(\text{PARIS}_n^1) = 2 \cdot n + 3$ . From this and Proposition 2, it follows immediately that  $\text{PAR} \in \text{IS}_{\text{br}}^1 \setminus B(2 \cdot n + 3)$ .  $\square$

Theorems 1 and 2 give rise to the question whether  $\text{PAR} \notin \text{IS}_{\text{br}}^0 \setminus B(2 \cdot n + 3)$ . In Section 4, this question will be answered in the affirmative. It is still an open question whether there exist a  $k \geq 1$  and an  $f : \mathbb{N} \rightarrow \mathbb{N}$  with  $f(n) < 2 \cdot n + 3$  for all  $n > 0$  such that  $\text{PAR} \in \text{IS}_{\text{br}}^k \setminus B(f(n))$ .

According to the view taken in [16], differences in the number of auxiliary Boolean registers whose use contributes to computing the function at hand always go with algorithmic differences. This view is supported by the instruction sequences  $\text{PARIS}_n^0$  and  $\text{PARIS}_n^1$ : in addition to having different lengths, they express undeniably quite different algorithms to compute  $\text{PAR}_n$  (for  $n > 1$ ).

## 4 Shorter Instruction Sequences with Auxiliary Registers

In Section 3, we have described how the parity functions can be computed by instruction sequences without the use of auxiliary Boolean registers and with the use of auxiliary Boolean registers. In the current section, we show that the smaller lengths of the instruction sequences in the latter case cannot be obtained without the use of auxiliary Boolean registers. In other words, we show that  $\text{PAR} \notin \text{IS}_{\text{br}}^0 \setminus B(2 \cdot n + 3)$ .

First, we introduce some notation that will be used in this section. We write  $\bar{b}$ , where  $b \in \{0, 1\}$ , for the complement of  $b$ , i.e.  $\bar{0} = 1$  and  $\bar{1} = 0$ ; we write  $\bar{f}$ , where  $f$  is an  $n$ -ary Boolean function, for the unique  $n$ -ary Boolean function  $g$  such that  $g(b_1, \dots, b_n) = \overline{f(b_1, \dots, b_n)}$ ; and we write  $\overline{F}$ , where  $F$  is a Boolean function family  $\langle f_n \rangle_{n \in \mathbb{N}}$ , for the Boolean function family  $\langle \bar{f}_n \rangle_{n \in \mathbb{N}}$ .

We know the following about the complexity of  $\text{PAR}$ .

**Proposition 3.** *For each  $k \in \mathbb{N}$ ,  $\text{PAR} \in \text{IS}_{\text{br}}^0 \setminus B(k)$  implies  $\overline{\text{PAR}} \in \text{IS}_{\text{br}}^0 \setminus B(k)$ .*

*Proof.* It is sufficient to prove for an arbitrary  $n \in \mathbb{N}_1$  that any instruction sequence from  $\text{IS}_{\text{br}}^0$  that computes  $\text{PAR}_n$  can be transformed into one with the same length that computes  $\overline{\text{PAR}_n}$ . So let  $n \in \mathbb{N}_1$ , and let  $X \in \text{IS}_{\text{br}}^0$  be such that  $X$  computes  $\text{PAR}_n$ . We distinguish two cases:  $n$  is odd and  $n$  is even.

If  $n$  is odd, then  $\overline{\text{PAR}_n}(b_1, \dots, b_n) = \text{PAR}_n(\bar{b}_1, \dots, \bar{b}_n)$ . This implies that  $\text{PAR}_n$  is computed by the instruction sequence from  $\text{IS}_{\text{br}}^0$  obtained from  $X$  by replacing, for each  $i \in \{1, \dots, n\}$ ,  $+\text{in}:i.\text{get}$  by  $-\text{in}:i.\text{get}$  and vice versa.

If  $n$  is even, then  $\overline{\text{PAR}_n}(b_1, \dots, b_n) = \text{PAR}_n(\bar{b}_1, b_2, \dots, b_n)$ . This implies that  $\text{PAR}_n$  is computed by the instruction sequence from  $\text{IS}_{\text{br}}^0$  obtained from  $X$  by replacing  $+\text{in}:1.\text{get}$  by  $-\text{in}:1.\text{get}$  and vice versa.  $\square$

The following three lemmas bring us step by step to the main result of this section, namely  $\text{PAR} \notin \text{IS}_{\text{br}}^0 \setminus B(2 \cdot n + 3)$ .

**Lemma 1.** *Let  $X \in \text{IS}_{\text{br}}^0$  be such that  $X$  computes  $\text{PAR}_2$ . Then  $\text{length}(X) \geq 6$ .*

*Proof.* Let  $X \in \text{IS}_{\text{br}}^0$  be such that  $X$  computes  $\text{PAR}_2$ . The following observations can be made about  $X$ :

- (1) for one  $i \in \{1, 2\}$ , there must be at least one instruction of the form  $+\text{in}:i.\text{get}$  or the form  $-\text{in}:i.\text{get}$  in  $X$  and, for the other  $i \in \{1, 2\}$ , there must be at least two instructions of the form  $+\text{in}:i.\text{get}$  or the form  $-\text{in}:i.\text{get}$  in  $X$  — because otherwise the final content of **out** will not in all cases be dependent on the content of both **in:1** and **in:2**;
- (2) there must be at least one occurrence of **out.set:1** in  $X$  and the last occurrence of **out.set:1** in  $X$  must precede an occurrence of **!** — because otherwise the final content of **out** will never be 1;
- (3) there must be at least two occurrences of **!** in  $X$  unless there occurs an instruction of the form  $\#l$  in  $X$  whose effect is that the last occurrence of **out.set:1** in  $X$  is skipped — because otherwise the final content of **out** will never be 0.



It follows immediately from these observations that  $\text{length}(X) \geq 6$ .  $\square$

In the proofs of the next two lemmas, we use the term *test on in:i* to refer to an instruction of the form  $+\text{in}:i.\text{get}$  or the form  $-\text{in}:i.\text{get}$ , and we use the term *test* to refer to an instruction that is a test on in:1 or a test on in:2.

**Lemma 2.** *Let  $X \in \text{IS}_{\text{br}}^0$  be such that  $X$  computes  $\text{PAR}_2$ . Then  $\text{length}(X) \geq 7$ .*

*Proof.* Let  $X \in \text{IS}_{\text{br}}^0$  be such that  $X$  computes  $\text{PAR}_2$ . Then, by Lemma 1,  $\text{length}(X) \geq 6$ . It remains to be proved that  $\text{length}(X) \neq 6$ . Below this is proved by contradiction, using the following assumption with respect to  $X$ :

- there does not occur an instruction of the form  $\#l$  in  $X$  whose effect is that the last occurrence of  $\text{out.set}:1$  in  $X$  is skipped.

This assumption can be made without loss of generality because, if it is not met,  $X$  can be replaced by an instruction sequence from  $\text{IS}_{\text{br}}^0$  of the same or smaller length by which it is met.

Assume that  $\text{length}(X) = 6$ , and suppose that  $X = u_1 ; \dots ; u_6$ . The first occurrence of  $!$  must be preceded by at least one test on in:1 and one test on in:2, because otherwise the final content of  $\text{out}$  will in some cases not be dependent on the content of both in:1 and in:2. From this and observations (1), (2), and (3) from the proof of Lemma 1, it follows that either  $u_3$  or  $u_4$  must be  $!$  and that  $u_5$  must be  $\text{out.set}:1$  and  $u_6$  must be  $!$ . However, if  $u_3 \equiv !$ , then termination can take place after performing only one test. Because this means that the final content of  $\text{out}$  will still in some cases not be dependent on the content of both in:1 and in:2, it is impossible that  $u_3 \equiv !$ . So,  $X = u_1 ; u_2 ; u_3 ; ! ; \text{out.set}:1 ; !$ . Let  $Y = u_1 ; u_2 ; u_3 ; \text{out.set}:1 ; !$ . Then  $\text{length}(Y) = 5$  and, because  $X$  computes  $\text{PAR}_2$  and  $u_3$  is a test by observation (1) from the proof of Lemma 1,  $Y$  computes  $\text{PAR}_2$ . Hence, by Proposition 3, there exists a  $Z \in \text{IS}_{\text{br}}^0$  that computes  $\text{PAR}_2$  such that  $\text{length}(Z) \leq 5$ . This contradicts Lemma 1.  $\square$

Lemma 1 is used in the proof of Lemma 2. Lemma 2, in its turn, is used in a similar way below in the proof of Lemma 3.

A remark in advance about the proof of Lemma 3 is perhaps in order. A proof of this lemma needs basically an extremely extensive case distinction.<sup>3</sup> In the proof given below, the extent of the case distinction is strongly reduced by using various apposite properties of the instruction sequences concerned. The reduction, which is practically necessary, has led to a proof that might make the impression to be unstructured.

**Lemma 3.** *Let  $X \in \text{IS}_{\text{br}}^0$  be such that  $X$  computes  $\text{PAR}_2$ . Then  $\text{length}(X) > 7$ .*

*Proof.* Let  $X \in \text{IS}_{\text{br}}^0$  be such that  $X$  computes  $\text{PAR}_2$ . Then, by Lemma 2,  $\text{length}(X) \geq 7$ . It remains to be proved that  $\text{length}(X) \neq 7$ . Below this is proved by contradiction, using the following assumptions with respect to  $X$ :

---

<sup>3</sup> At bottom, there are in the order of  $10^7$  different instruction sequences to consider. By the assumptions made at the beginning of the proof of the lemma this number can be reduced by a factor of 10.

- the first instruction of  $X$  is a test;
- the instructions  $\#0$  and  $\#1$  do not occur in  $X$ ;
- there does not occur an instruction of the form  $\#l$  in  $X$  whose effect is that the last occurrence of  $\text{out.set:1}$  in  $X$  is skipped.

These assumptions can be made without loss of generality because, for each of them, if it is not met,  $X$  can be replaced by an instruction sequence from  $\text{IS}_{\text{br}}^0$  of the same or smaller length by which it is met.

Assume that  $\text{length}(X) = 7$ , and suppose that  $X = u_1 ; \dots ; u_7$ . From observations (1), (2), and (3) from the proof of Lemma 1, it follows that either  $u_5$  must be  $\text{out.set:1}$  and  $u_6$  must be  $!$  or  $u_6$  must be  $\text{out.set:1}$  and  $u_7$  must be  $!$ . However, in the former case  $X$  can be replaced by a shorter instruction sequence from  $\text{IS}_{\text{br}}^0$ . Because this contradicts Lemma 2, it is impossible that  $u_5 \equiv \text{out.set:1}$  and  $u_6 \equiv !$ . So  $X = u_1 ; \dots ; u_5 ; \text{out.set:1} ; !$ .

Consider the case that  $u_1 \equiv +\text{in:1.get}$ . Because the first occurrence of  $!$  must be preceded by at least one test on  $\text{in:1}$  and one test on  $\text{in:2}$ ,  $u_2$  must be either  $\#2$ ,  $\#3$ ,  $\#4$  or a test. If  $u_2 \equiv \#2$ , then  $X$  can be replaced by an instruction sequence whose length is 6, to wit  $-\text{in:1.get} ; u_3 ; u_4 ; u_5 ; \text{out.set:1} ; !$ . Because this contradicts Lemma 2, it is impossible that  $u_2 \equiv \#2$ . If  $u_2 \equiv \#4$  and moreover  $\text{in:1}$  contains 1, then the final content of  $\text{out}$  will not be dependent on the content of  $\text{in:2}$ . Therefore, it is also impossible that  $u_2 \equiv \#4$ . The cases that  $u_2 \equiv \#3$  and  $u_2$  is a test need more extensive investigation.

Because the first occurrence of  $!$  must be preceded by at least one test on  $\text{in:1}$  and one test on  $\text{in:1}$ ,  $u_3$  must be either  $\#2$ ,  $\#3$  or a test if  $u_2 \equiv \#3$ . If  $u_2 \equiv \#3$  and  $u_3 \equiv \#2$ , then  $X$  can be replaced by an instruction sequence whose length is 4, to wit  $u_4 ; u_5 ; \text{out.set:1} ; !$ . Because this contradicts Lemma 2, it is impossible that  $u_3 \equiv \#2$  if  $u_2 \equiv \#3$ . If  $u_2 \equiv \#3$  and  $u_3 \equiv \#3$  and moreover  $\text{in:1}$  contains 0, then the final content of  $\text{out}$  will not be dependent on the content of  $\text{in:2}$ . Therefore, it is impossible that  $u_3 \equiv \#3$  if  $u_2 \equiv \#3$ . So  $u_3$  must be a test if  $u_2 \equiv \#3$ . Because  $\text{out.set:1}$  has to be executed if  $\text{in:1}$  contains 1 and  $\text{in:2}$  contains 0,  $u_5 \equiv -\text{in:2.get}$  if  $u_2 \equiv \#3$ . Moreover, because there must be at least two occurrences of  $!$  in  $X$  and  $u_3$  must be a test if  $u_2 \equiv \#3$ ,  $u_4$  must be  $!$  if  $u_2 \equiv \#3$ . So  $X = +\text{in:1.get} ; \#3 ; u_3 ; ! ; -\text{in:2.get} ; \text{out.set:1} ; !$  and  $u_3$  must be a test if  $u_2 \equiv \#3$ . In the case that  $u_2 \equiv \#3$ , the subcase that  $u_3$  is a test needs more extensive investigation.

If  $u_3$  is a test, it is either  $+\text{in:1.get}$ ,  $-\text{in:1.get}$ ,  $+\text{in:2.get}$  or  $-\text{in:2.get}$ . If  $u_2 \equiv \#3$  and  $u_3 \equiv +\text{in:1.get}$ , then the final content of  $\text{out}$  will be independent of the content of  $\text{in:1}$ . If  $u_2 \equiv \#3$  and  $u_3 \equiv -\text{in:1.get}$ , then the final content of  $\text{out}$  will be independent of the content of  $\text{in:2}$  if  $\text{in:1}$  contains 0. If  $u_2 \equiv \#3$  and either  $u_3 \equiv +\text{in:2.get}$  or  $u_3 \equiv -\text{in:2.get}$ , then the final content of  $\text{out}$  will be wrong if  $\text{in:1}$  contains 0 and  $\text{in:2}$  contains 1. Therefore, it is impossible that  $u_3$  is a test if  $u_2 \equiv \#3$ . Because there are no more alternatives left for  $u_3$  if  $u_2 \equiv \#3$ , it is impossible that  $u_2 \equiv \#3$ . The left-over case for  $u_2$  is the case that  $u_2$  is a test. This case needs very extensive investigation.

If  $u_2$  is a test, it is either  $+\text{in:1.get}$ ,  $-\text{in:1.get}$ ,  $+\text{in:2.get}$  or  $-\text{in:2.get}$ . If  $u_2 \equiv +\text{in:1.get}$ , then  $X$  can be replaced by an instruction sequence whose length is 5,

to wit  $u_3; u_4; u_5; \text{out.set:1}; !$ . Because this contradicts Lemma 2, it is impossible that  $u_2 \equiv +\text{in:1.get}$ . If  $u_2 \equiv -\text{in:1.get}$ , then  $X$  can be replaced by an instruction sequence whose length is 6, to wit  $-\text{in:1.get}; u_3; u_4; u_5; \text{out.set:1}; !$ . Because this contradicts Lemma 2, it is impossible that  $u_2 \equiv -\text{in:1.get}$ . The specific cases that  $u_2 \equiv +\text{in:2.get}$  and  $u_2 \equiv -\text{in:2.get}$  need more extensive investigation.

The following will be used in both these cases. If both  $u_2$  and  $u_3$  are tests, then either  $u_4$  or  $u_5$  must be  $!$  by observation (3) from the proof of Lemma 1. If  $u_5 \equiv !$ , then (a) it is impossible that  $u_4 \equiv !$ , because otherwise the final content of **out** will be independent of the content of **in:2** if **in:1** contains 0, and (b) it is impossible that  $u_4 \equiv \#2$  or  $u_4$  is a test, because otherwise  $X$  can be replaced by an instruction sequence whose length is 6 and this contradicts Lemma 2. So it is impossible that  $u_5 \equiv !$  if both  $u_2$  and  $u_3$  are tests and  $u_4$  must be  $!$  if both  $u_2$  and  $u_3$  are tests. Moreover, if  $u_2$  is a test, then it is impossible that  $u_3 \equiv !$ , because otherwise the final content of **out** will be independent of the content of **in:2** if **in:1** contains 0.

Because it is impossible that  $u_3 \equiv !$  if  $u_2$  is a test,  $u_3$  must be either  $\#2$ ,  $\#3$  or a test if  $u_2 \equiv +\text{in:2.get}$ . If  $u_2 \equiv +\text{in:2.get}$  and  $u_3 \equiv \#2$ , then the final content of **out** will be wrong if **in:1** contains 0 unless  $u_5 \equiv +\text{in:2.get}$ . However, then  $u_4$  must be  $!$  by observation (3) from the proof of Lemma 1 and the final content of **out** will be wrong if **in:1** contains 1 and **in:2** contains 1. Therefore, it is impossible that  $u_3 \equiv \#2$  if  $u_2 \equiv +\text{in:2.get}$ . If  $u_2 \equiv +\text{in:2.get}$  and  $u_3 \equiv \#3$ , then the final content of **out** will be independent of the content of **in:2** if **in:1** contains 0. Therefore, it is impossible that  $u_3 \equiv \#3$  if  $u_2 \equiv +\text{in:2.get}$ . So  $u_3$  must be a test if  $u_2 \equiv +\text{in:2.get}$ . We know that  $u_4$  must be  $!$  and it is impossible that  $u_5 \equiv !$  if both  $u_2$  and  $u_3$  are tests. This implies that, if  $u_2 \equiv +\text{in:2.get}$  and  $u_3$  is a test, the final content of **out** will be wrong if **in:1** contains 1 and **in:2** contains 0. Therefore, it is impossible that  $u_3$  is a test if  $u_2 \equiv +\text{in:2.get}$ . Because there are no more alternatives left for  $u_3$  if  $u_2 \equiv +\text{in:2.get}$ , it is impossible that  $u_2 \equiv +\text{in:2.get}$ .

Because it is impossible that  $u_3 \equiv !$  if  $u_2$  is a test,  $u_3$  must be either  $\#2$ ,  $\#3$  or a test if  $u_2 \equiv -\text{in:2.get}$ . If  $u_2 \equiv -\text{in:2.get}$  and  $u_3 \equiv \#2$ , then the final content of **out** will be wrong if **in:1** contains 1 and **in:2** contains 0 unless  $u_5 \equiv -\text{in:2.get}$ . However, then  $u_4$  must be  $!$  by observation (3) from the proof of Lemma 1 and the final content of **out** will be wrong if **in:1** contains 0 and **in:2** contains 1. Therefore, it is impossible that  $u_3 \equiv \#2$  if  $u_2 \equiv -\text{in:2.get}$ . If  $u_2 \equiv -\text{in:2.get}$  and  $u_3 \equiv \#3$ , then the final content of **out** will be independent of the content of **in:2** if **in:1** contains 0. Therefore, it is impossible that  $u_3 \equiv \#3$  if  $u_2 \equiv -\text{in:2.get}$ . So  $u_3$  must be a test if  $u_2 \equiv -\text{in:2.get}$ . In the case that  $u_2 \equiv -\text{in:2.get}$ , the subcase that  $u_3$  is a test needs more extensive investigation.

If  $u_3$  is a test, it is either  $+\text{in:1.get}$ ,  $-\text{in:1.get}$ ,  $+\text{in:2.get}$  or  $-\text{in:2.get}$ . We know that  $u_4$  must be  $!$  and it is impossible that  $u_5 \equiv !$  if both  $u_2$  and  $u_3$  are tests. This implies that (a) if  $u_2 \equiv -\text{in:2.get}$  and either  $u_3 \equiv +\text{in:1.get}$  or  $u_3 \equiv -\text{in:2.get}$ , then the final content of **out** will be wrong if **in:1** contains 1 and **in:2** contains 0 and (b) if  $u_2 \equiv -\text{in:2.get}$  and either  $u_3 \equiv -\text{in:1.get}$  or  $u_3 \equiv +\text{in:2.get}$ , then the final content of **out** will be wrong if **in:1** contains 0 and **in:2** contains 1. Because

there are no more alternatives left for  $u_3$  if  $u_2 \equiv -\text{in}:2.\text{get}$ , it is impossible that  $u_2 \equiv -\text{in}:2.\text{get}$ .

Because there are no more alternatives left for  $u_2$ , it is impossible that  $u_1 \equiv +\text{in}:1.\text{get}$ . Analogously, we find that it is impossible that  $u_1 \equiv -\text{in}:1.\text{get}$ , it is impossible that  $u_1 \equiv +\text{in}:2.\text{get}$ , and it is impossible that  $u_1 \equiv -\text{in}:2.\text{get}$ . Hence, it is impossible that  $\text{length}(X) = 7$ .  $\square$

Lemma 2 is used in the proof of Lemma 3. The latter lemma, in its turn, is used below in the proof of the main result of this section.

**Theorem 3.**  $\text{PAR} \notin \text{IS}_{\text{br}}^0 \setminus B(2 \cdot n + 3)$ .

*Proof.* It is sufficient to prove that there exists an  $m \in \mathbb{N}_1$  such that, for all  $n \geq m$ , for all instruction sequences  $X \in \text{IS}_{\text{br}}^0$  that compute  $\text{PAR}_n$ ,  $\text{length}(X) > 2 \cdot n + 3$ . We take  $m = 2$  because of Lemma 3 and prove the property to be proved for all  $n \geq 2$  by induction on  $n$ . The basis step consists of proving that, for all instruction sequences  $X \in \text{IS}_{\text{br}}^0$  that compute  $\text{PAR}_2$ ,  $\text{length}(X) > 7$ . This follows trivially from Lemma 3. The inductive step is proved below by contradiction.

Suppose that  $X \in \text{IS}_{\text{br}}^0$ ,  $X$  computes  $\text{PAR}_{n+1}$ , and  $\text{length}(X) \leq 2 \cdot (n+1) + 3$ . Without loss of generality, we may assume that there does not exist an  $X' \in \text{IS}_{\text{br}}^0$  that computes  $\text{PAR}_{n+1}$  such that  $\text{length}(X') < \text{length}(X)$ . Therefore, we may assume that  $X = u_1 ; \dots ; u_k$  ( $k \leq 2 \cdot (n+1) + 3$ ) where  $u_1 \equiv +\text{in}:n+1.\text{get}$  or  $u_1 \equiv -\text{in}:n+1.\text{get}$ . We distinguish these two cases.

In the case that  $u_1 \equiv -\text{in}:n+1.\text{get}$ , we consider the case that  $\text{in}:n+1$  contains 1. In this case, after execution of  $u_1$ , execution proceeds with  $u_3$ . Let  $Y \in \text{IS}_{\text{br}}^0$  be obtained from  $u_3 ; \dots ; u_k$  by first replacing  $+\text{in}:n+1.\text{get}$  by  $\#1$  and  $-\text{in}:n+1.\text{get}$  by  $\#2$  and then removing the  $\#1$ 's. Then, we have that  $Y$  computes  $\overline{\text{PAR}}_n$  and  $\text{length}(Y) \leq \text{length}(X) - 2 \leq 2 \cdot n + 3$ . Hence, by Proposition 3, there exists a  $Z \in \text{IS}_{\text{br}}^0$  that computes  $\text{PAR}_n$  such that  $\text{length}(Z) \leq 2 \cdot n + 3$ . This contradicts the induction hypothesis.

In the case that  $u_1 \equiv +\text{in}:n+1.\text{get}$ , we consider the case that  $\text{in}:n+1$  contains 0. This case leads to a contradiction in the same way as above, but without the use of Proposition 3.  $\square$

The following is a corollary of Theorems 2 and 3.

**Corollary 1.**  $\text{IS}_{\text{br}}^0 \setminus B(2 \cdot n + 3) \subset \text{IS}_{\text{br}}^1 \setminus B(2 \cdot n + 3)$ .<sup>4</sup>

## 5 Concluding Remarks

We have shown, in a setting where programs are instruction sequences acting on Boolean registers, that, in the case of the parity functions, smaller programs are possible with the use of one auxiliary Boolean register than without the use of an auxiliary Boolean register. This result supports a basic intuition behind the storage of auxiliary data, namely the intuition that this makes possible a reduction of the size of a program.

<sup>4</sup> As usual, the symbol  $\subset$  is used to denote the proper inclusion relation.

Of course, more results supporting this intuition would be nice. Adversely, we do not even know at the present time how to prove, for example, that there exists a Boolean function family for which smaller instruction sequences are possible with the use of two auxiliary Boolean registers than with the use of one auxiliary Boolean register. Moreover, we do not know of results in other theoretical settings that support the intuition that the storage of auxiliary data makes possible a reduction of the size of a program.

It is still an open question whether smaller programs are possible with the use of one auxiliary Boolean register than without the use of an auxiliary Boolean register in the absence of instructions to complement the content of auxiliary Boolean registers. We conjecture that this question can be answered in the affirmative, but the practical problem is that the number of different instruction sequences to be considered in the proof is increased by a factor of  $10^4$ .

It is intuitively clear that the instructions sequences  $\text{PARIS}_n^0$  and  $\text{PARIS}_n^1$  from Section 3 express parameterized algorithms. However, it is very difficult to develop a precise viewpoint on what is a parameterized algorithm. In [16], we looked for an equivalence relation on instruction sequences that captures to a reasonable degree the intuitive notion that two instruction sequences express the same algorithm. In that paper, we considered non-parameterized algorithms only because it turned out to be already very difficult to develop a precise viewpoint on what is a non-parameterized algorithm.

## Acknowledgements

We thank two anonymous referees for carefully reading a preliminary version of this paper and for suggesting improvements of the presentation of the paper.

## References

1. Bergstra JA, Middelburg CA. Instruction Sequence Based Non-uniform Complexity Classes. *Scientific Annals of Computer Science*. 2014;24(1):47–89. doi:10.7561/SACS.2014.1.47.
2. Salomon D. Coding for Data and Computer Communications. Berlin: Springer-Verlag; 2005. doi:10.1007/b102531.
3. Furst M, Saxe JB, Sipser M. Parity, Circuits, and the Polynomial-Time Hierarchy. *Mathematical Systems Theory*. 1984;17(1):13–27. doi:10.1007/BF01744431.
4. Håstad J. Almost Optimal Lower Bounds for Small Depth Circuits. In: *STOC '86*. ACM Press; 1986. p. 6–20. doi:10.1145/12130.12132.
5. Wegener I. The Complexity of the Parity Function in Unbounded Fan-In, Unbounded Depth Circuits. *Theoretical Computer Science*. 1991;85(1):155–170. doi:10.1016/0304-3975(91)90052-4.
6. Beame P, Impagliazzo R, Srinivasan S. Approximating  $\text{AC}^0$  by Small Height Decision Trees and a Deterministic Algorithm for  $\#\text{AC}^0\text{SAT}$ . In: *CCC 2012*. IEEE Computer Society Press; 2012. p. 117–125. doi:10.1109/CCC.2012.40.
7. Impagliazzo R, Matthews W, Paturi R. A Satisfiability Algorithm for  $\text{AC}^0$ . In: *SODA 2012*. SIAM; 2012. p. 961–972. doi:10.1137/1.9781611973099.77.

8. Håstad J. On the Correlation of Parity and Small-Depth Circuits. *SIAM Journal of Computing*. 2014;43(5):1699–1708. doi:10.1137/120897432.
9. Bergstra JA, Middelburg CA. On Instruction Sets for Boolean Registers in Program Algebra. *CoRR*; 2015. arXiv:1502.00238 [cs.PL]. Accepted for publication by *Scientific Annals of Computer Science*.
10. Böhm C, Jacopini G. Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules. *Communications of the ACM*. 1966;9(5):366–371. doi:10.1145/355592.365646.
11. Cooper DC. Böhm and Jacopini’s Reduction of Flow Charts. *Communications of the ACM*. 1967;10(8):463, 473. doi:10.1145/363534.363539.
12. Owicki S, Gries D. An Axiomatic Proof Technique for Parallel Programs I. *Acta Informatica*. 1976;6(4):319–340. doi:10.1007/BF00268134.
13. Apt KR, de Boer FS, Olderog ER. *Verification of Sequential and Concurrent Programs*. 3rd ed. *Texts in Computer Science*. Berlin: Springer-Verlag; 2009. doi:10.1007/978-1-84882-745-5.
14. Bergstra JA, Loots ME. Program Algebra for Sequential Code. *Journal of Logic and Algebraic Programming*. 2002;51(2):125–156. doi:10.1016/S1567-8326(02)00018-8.
15. Bergstra JA, Middelburg CA. *Instruction Sequences for Computer Science*. vol. 2 of *Atlantis Studies in Computing*. Amsterdam: Atlantis Press; 2012. doi:10.2991/978-94-91216-65-7.
16. Bergstra JA, Middelburg CA. On Algorithmic Equivalence of Instruction Sequences for Computing Bit String Functions. *Fundamenta Informaticae*. 2015;138(4):411–434. doi:10.3233/FI-2015-1219.